In [1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
import heapq
```

Greedy Search

In [2]:
```
map_resolution = 0.5
angular_resolution = 5
num_ori = 360 / angular_resolution
```

In [3]:
```
data = loadmat('cspaceMaps_python.mat')
angles = data['angles'][0]
map = data['mapMatrix']
cspace = {}
for index, angle in enumerate(angles):
    space = data['cspace'][0][index]
    cspace[angle] = space
```

In [4]:
```
class Node:
    def __init__(self, x, y, theta, g=0, h=0, f=0, parent=None):
        self.x = x
        self.y = y
        self.theta = theta % 360
        self.g = g
        self.h = h
        self.f = f
        self.parent = parent
    def __eq__(self, other):
        return (self.x == other.x and self.y == other.y and self.theta == other.

    def __lt__(self, other):
        return self.f < other.f

    def key(self, angular_resolution):
        return f"{self.x}_{self.y}_{int(self.theta / angular_resolution)}"
```

In [9]:
```
class A_Star:
    def __init__(self, start, goal, cspace_maps, angles,map_resolution=0.5, angu
        self.start = start
        self.goal = goal
        self.cspace_maps = cspace_maps
        self.map_resolution = map_resolution
        self.angular_resolution = angular_resolution
        self.open_list = []
        self.closed_set = set()
        self.map_size = cspace_maps[0].shape
        self.path = []
        self.total_steps = 0
        self.angles = angles

    def init_open_list(self):
        start_node = Node(self.start[0], self.start[1], self.start[2])
        start_node.h = self.get_heuristic(start_node)
```

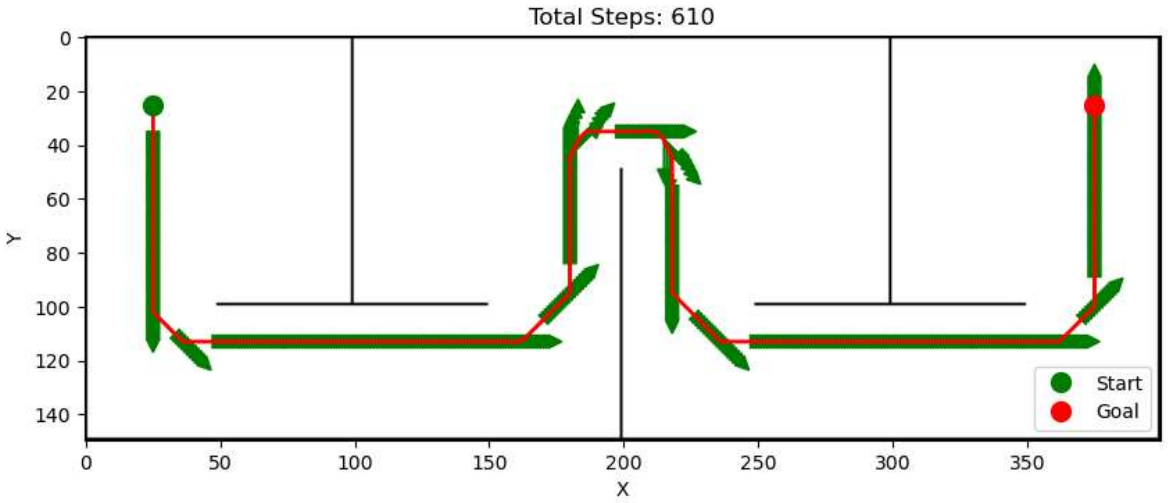
```
        start_node.f = start_node.g + start_node.h
        heapq.heappush(self.open_list, (start_node.f, start_node))

    def is_goal(self,node):
        return node.x == self.goal[0] and node.y == self.goal[1]

    def movement_cost(self, node, neighbor):
        step_cost = 0
        # Translation Cost
        if node.x != neighbor.x or node.y != neighbor.y:
            step_cost += 1
        # Rotation Cost
        else:
            angle_diff = min(abs(node.theta - neighbor.theta), 360 - abs(node.th
            step_cost += angle_diff / self.angular_resolution
        return step_cost

    def get_heuristic(self, node):
        '''
        Apply Euclidean Distance to a node and goal.
        Args:
            node: input node

        Returns: heuristic value
        '''
        # Position Diff, Euclidean Distance
        pos_diff = np.sqrt((node.x - self.goal[0]) ** 2 + (node.y - self.goal[1]
        # Angle Diff
        angle_diff = min(abs(node.theta - self.goal[2]), 360 - abs(node.theta -
        return pos_diff + angle_diff

    def reconstruct_path(self,node):
        path = []
        total_steps = node.g
        while node:
            path.append((node.x, node.y, node.theta))
            node = node.parent
        path.reverse()
        return path, total_steps

    def gen_neighbors(self, node):
        neighbors = []
        directions = [
            (0, 1, 90),
            (1, 0, 0),
            (0, -1, 270),
            (-1, 0, 180),
            (1, 1, 45),
            (1, -1, 315),
            (-1, -1, 225),
            (-1, 1, 135),
        ]
        for dx, dy, target_theta in directions:
            angle_diff = (target_theta - node.theta + 360) % 360
            steps_to_rotate = angle_diff / self.angular_resolution
            # Need Rotation
            if steps_to_rotate != 0:
                rotation_direction = self.angular_resolution if angle_diff <= 18
                temp_theta = node.theta
                collision = False
```

```
                # Gradually make rotation & Check collision: 5 degree at a time
                while temp_theta != target_theta:
                    temp_theta = (temp_theta + rotation_direction + 360) % 360
                    angle_idx = int(temp_theta / self.angular_resolution) % len(
                    if self.cspace_maps[self.angles[angle_idx]][node.y][node.x]
                        collision = True
                        break
                if collision:
                    continue

            # Check boundary
            new_x = node.x + dx
            new_y = node.y + dy
            if new_x < 0 or new_x >= self.map_size[1] or new_y < 0 or new_y >= s
                continue

            # Check collision with new orientation
            angle_idx = int(target_theta / self.angular_resolution) % len(self.c
            if self.cspace_maps[self.angles[angle_idx]][new_y][new_x] > 0:
                continue
            neighbor = Node(new_x, new_y, target_theta)
            neighbors.append(neighbor)
        return neighbors

    def run(self):
        # Open List is not empty
        while self.open_list:
            current_node = heapq.heappop(self.open_list)[1]
            # Check Goal
            if self.is_goal(current_node):
                self.path, self.total_steps = self.reconstruct_path(current_node
                return
            # Add current node to closed set
            key = current_node.key(self.angular_resolution)
            self.closed_set.add(key)

            # Get Neighbor
            neighbors = self.gen_neighbors(current_node)
            for neighbor in neighbors:
                # Check if neighbor in closed list
                key = neighbor.key(self.angular_resolution)
                if key in self.closed_set:
                    continue

                # Compute g for neighbor
                tentative_g = current_node.g + self.movement_cost(current_node,

                # Update if neighbor in open list
                in_open_list = False
                for item in self.open_list:
                    if neighbor == item[1]:
                        in_open_list = True
                        if tentative_g < neighbor.g:
                            neighbor.g = tentative_g
                            neighbor.f = neighbor.g + neighbor.h
                            neighbor.parent = current_node
                        break
                # Add if neighbor not in open list
                if not in_open_list:
                    neighbor.g = tentative_g
```

```
                    neighbor.h = self.get_heuristic(neighbor)
                    neighbor.f = neighbor.g + neighbor.h
                    neighbor.parent = current_node
                    heapq.heappush(self.open_list, (neighbor.f, neighbor))
        return
```

In [10]:
```
start = [25,25,90]
goal = [375,25,0]
map_resolution = 0.5
angular_resolution = 5
planner = A_Star(start,goal,cspace,angles,map_resolution,angular_resolution)
planner.init_open_list()
planner.run()
```

In [11]:
```
path = planner.path
path_x = [p[0] for p in path]
path_y = [p[1] for p in path]
path_theta = [p[2] for p in path]
map_display = 1 - map
plt.figure(figsize=(10, 8))
plt.imshow(map_display, cmap='gray', origin='upper')
plt.plot(path_x, path_y, 'r-', linewidth=2)

plt.plot(start[0], start[1], 'go', markersize=10, label='Start')
plt.plot(goal[0], goal[1], 'ro', markersize=10, label='Goal')
for x, y, theta in path:
    dx = np.cos(np.radians(theta)) * 10
    dy = np.sin(np.radians(theta)) * 10
    plt.arrow(x, y, dx, dy, head_width=5, head_length=5, fc='green', ec='green')
plt.title(f'Total Steps: {planner.total_steps}')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
#plt.savefig('greedy.png')
plt.show()
```



In [ ]:

In [1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
import heapq
from scipy.ndimage import distance_transform_edt
from skimage.morphology import skeletonize
from skimage.draw import line
```

Voronoi Diagram

In [20]:
```
map_resolution = 0.5
angular_resolution = 5
num_ori = 360 / angular_resolution
```

In [21]:
```
data = loadmat('cspaceMaps_python.mat')
angles = data['angles'][0]
map = data['mapMatrix']
cspace = {}
for index, angle in enumerate(angles):
    space = data['cspace'][0][index]
    cspace[angle] = space
```

In [22]:
```
class Voronoi():
    def __init__(self, start, goal, cspace_map):
        self.start = start
        self.goal = goal
        self.cspace = cspace_map
        self.skeleton_set = None

    def connect_to_skeleton(self, pos):
        nodes_array = np.array(list(self.skeleton_set))
        distances = np.linalg.norm(nodes_array - pos, axis=1)
        nearest_idx = np.argmin(distances)
        nearest_skel_point = tuple(nodes_array[nearest_idx])
        rr, cc = line(int(pos[1]), int(pos[0]), int(nearest_skel_point[1]), int(
        # Check collision
        for y, x in zip(rr, cc):
            if self.cspace[y, x] != 0:
                return None
        path_points = list(zip(cc, rr))
        return path_points

    def add_start_goal(self, start, goal):
        start_pos = (start[0], start[1])
        goal_pos = (goal[0], goal[1])
        start_path = self.connect_to_skeleton(start_pos)
        if start_path:
            self.skeleton_set.update(start_path)
        goal_path = self.connect_to_skeleton(goal_pos)
        if goal_path:
            self.skeleton_set.update(goal_path)


    def run(self):
```

```
        # build voronoi based on distance map
        distance_map = distance_transform_edt(self.cspace == 0)
        binary_map = distance_map > 0
        skeleton = skeletonize(binary_map)
        skeleton_indices = np.where(skeleton)
        skeleton_points = np.column_stack((skeleton_indices[1], skeleton_indices
        self.skeleton_set = set((int(x), int(y)) for x, y in skeleton_points)
        # Add start and goal
        self.add_start_goal(self.start, self.goal)
        return self.skeleton_set
```

In [23]:
```
class Node:
    def __init__(self, x, y, theta, g=0, h=0, f=0, parent=None):
        self.x = x
        self.y = y
        self.theta = theta % 360
        self.g = g
        self.h = h
        self.f = f
        self.parent = parent
    def __eq__(self, other):
        return (self.x == other.x and self.y == other.y and self.theta == other.

    def __lt__(self, other):
        return self.f < other.f

    def key(self, angular_resolution):
        return f"{self.x}_{self.y}_{int(self.theta / angular_resolution)}"
```

In [24]:
```
class A_Star:
    def __init__(self, start, goal, cspace_maps, map_resolution=0.5, angular_res
        self.start = start
        self.goal = goal
        self.cspace_maps = cspace_maps
        self.map_resolution = map_resolution
        self.angular_resolution = angular_resolution
        self.open_list = []
        self.closed_set = set()
        self.map_size = cspace_maps[0].shape
        self.path = []
        self.total_steps = 0
        self.skeleton_set = skeleton_set

    def init_open_list(self):
        start_node = Node(self.start[0], self.start[1], self.start[2])
        start_node.h = self.get_heuristic(start_node)
        start_node.f = start_node.g + start_node.h
        heapq.heappush(self.open_list, (start_node.f, start_node))

    def is_goal(self,node):
        return node.x == self.goal[0] and node.y == self.goal[1]

    def movement_cost(self, node, neighbor):
        step_cost = 0
        # Translation Cost
        if node.x != neighbor.x or node.y != neighbor.y:
            step_cost += 1
        # Rotation Cost
        '''angle_diff = min(abs(node.theta - neighbor.theta), 360 - abs(node.the
```

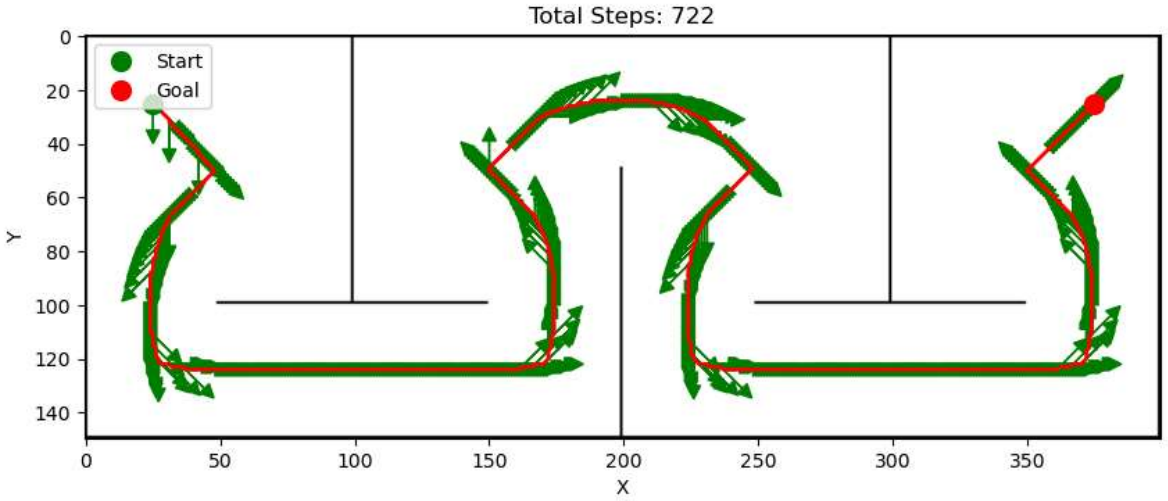
```
        step_cost += angle_diff / angular_resolution'''
        return step_cost

    def get_heuristic(self, node):
        '''
        Apply Euclidean distance to a node and goal.
        Args:
            node: input node

        Returns: heuristic value
        '''
        # Position Diff, Euclidean Distance
        pos_diff = np.sqrt((node.x - self.goal[0]) ** 2 + (node.y - self.goal[1]
        # Angle Diff
        angle_diff = min(abs(node.theta - self.goal[2]), 360 - abs(node.theta -
        return pos_diff + angle_diff

    def reconstruct_path(self,node):
        path = []
        total_steps = node.g
        while node:
            path.append((node.x, node.y, node.theta))
            node = node.parent
        path.reverse()
        return path, total_steps

    def gen_neighbors(self, node):
        neighbors = []
        directions = [
            (0, 1, 90),
            (1, 0, 0),
            (0, -1, 270),
            (-1, 0, 180),
            (1, 1, 45),
            (1, -1, 315),
            (-1, -1, 225),
            (-1, 1, 135),
        ]
        for dx, dy, target_theta in directions:
            new_x = node.x + dx
            new_y = node.y + dy

            # Not on Voronoi Diagram
            if self.skeleton_set and (new_x, new_y) not in self.skeleton_set:
                continue

            angle_diff = (target_theta - node.theta + 360) % 360
            steps_to_rotate = angle_diff / angular_resolution
            # Need Rotation
            if steps_to_rotate != 0:
                rotation_direction = angular_resolution if angle_diff <= 180 els
                temp_theta = node.theta
                collision = False
                # Gradually make rotation & Check collision: 5 degree at a time
                while temp_theta != target_theta:
                    temp_theta = (temp_theta + rotation_direction + 360) % 360
                    angle_idx = int(temp_theta / angular_resolution) % len(self.
                    if self.cspace_maps[angles[angle_idx]][node.y][node.x] > 0:
                        collision = True
                        break
```

```
                if collision:
                    continue

            # Check boundary
            if new_x < 0 or new_x >= self.map_size[1] or new_y < 0 or new_y >= s
                continue

            # Check collision with new orientation
            angle_idx = int(target_theta / angular_resolution) % len(self.cspace
            if self.cspace_maps[angles[angle_idx]][new_y][new_x] > 0:
                continue
            neighbor = Node(new_x, new_y, target_theta)
            neighbors.append(neighbor)
        return neighbors

    def run(self):
        # Open List is not empty
        while self.open_list:
            current_node = heapq.heappop(self.open_list)[1]
            # Check Goal
            if self.is_goal(current_node):
                self.path, self.total_steps = self.reconstruct_path(current_node
                return
            # Add current node to closed set
            key = current_node.key(self.angular_resolution)
            self.closed_set.add(key)

            # Get Neighbor
            neighbors = self.gen_neighbors(current_node)
            for neighbor in neighbors:
                # Check if neighbor in closed list
                key = neighbor.key(self.angular_resolution)
                if key in self.closed_set:
                    continue

                # Compute g for neighbor
                tentative_g = current_node.g + self.movement_cost(current_node,

                # Update if neighbor in open list
                in_open_list = False
                for item in self.open_list:
                    if neighbor == item[1]:
                        in_open_list = True
                        if tentative_g < neighbor.g:
                            neighbor.g = tentative_g
                            neighbor.f = neighbor.g + neighbor.h
                            neighbor.parent = current_node
                        break
                # Add if neighbor not in open list
                if not in_open_list:
                    neighbor.g = tentative_g
                    neighbor.h = self.get_heuristic(neighbor)
                    neighbor.f = neighbor.g + neighbor.h
                    neighbor.parent = current_node
                    heapq.heappush(self.open_list, (neighbor.f, neighbor))
        return
```

In [25]:
```
start = [25,25,90]
goal = [375,25,0]
cspace_map = cspace[45]
```

```
vor = Voronoi(start, goal, cspace_map)
skeleton_set = vor.run()
planner = A_Star(start,goal,cspace,map_resolution,angular_resolution, skeleton_s
planner.init_open_list()
planner.run()
```

In [27]:
```
path = planner.path
path_x = [p[0] for p in path]
path_y = [p[1] for p in path]
path_theta = [p[2] for p in path]
map_display = 1 - map
plt.figure(figsize=(10, 8))
plt.imshow(map_display, cmap='gray', origin='upper')
plt.plot(path_x, path_y, 'r-', linewidth=2)

plt.plot(start[0], start[1], 'go', markersize=10, label='Start')
plt.plot(goal[0], goal[1], 'ro', markersize=10, label='Goal')
for x, y, theta in path:
    dx = np.cos(np.radians(theta)) * 10
    dy = np.sin(np.radians(theta)) * 10
    plt.arrow(x, y, dx, dy, head_width=5, head_length=5, fc='green', ec='green')

plt.title(f'Total Steps: {planner.total_steps}')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('voronoi.png')
plt.show()
```



In [ ]:

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
import heapq
from skimage.draw import line
import random
```

PRM

In [24]:
```
class PRM():
    def __init__(self, start, goal, num_sample, num_neighbor, cspace_map,angles,
        self.num_sample = num_sample
        self.num_neighbor = num_neighbor
        self.cspace = cspace_map
        self.edges = set()
        self.vertices = []
        self.start = start
        self.goal = goal
        self.angular_resolution = angular_resolution
        self.angles = angles

    def check_collision_node(self, node):
        node_x = node[0]
        node_y = node[1]
        node_theta = node[2]
        if self.cspace[node_theta][node_x][node_y] != 0:
            return True
        return False

    def check_collision_edge(self, q_1, q_2):
        # Check rotation
        angle_diff = (int(q_1[2]) - int(q_2[2]) + 360) % 360
        steps_to_rotate = angle_diff / self.angular_resolution
        if steps_to_rotate != 0:
            rotation_direction = self.angular_resolution if angle_diff <= 180 el
            temp_theta = int(q_1[2])
            while temp_theta != int(q_2[2]):
                temp_theta = (temp_theta + rotation_direction + 360) % 360
                # Rotation from q_1 to q_2 is blocked
                if self.check_collision_node((q_1[0], q_1[1], temp_theta)):
                    return True

        # Check connection
        rr,cc = line(q_1[0], q_1[1], q_2[0], q_2[1])
        for row, col in zip(rr,cc):
            if self.check_collision_node((row, col, q_2[2])):
                return True

        return False

    def add_neighbors(self, nodes):
        q_1 = nodes[0]
        q_2 = nodes[1]
        # Check if different point
        pos_diff = abs(int(q_1[0]) - int(q_2[0])) + abs(int(q_1[1]) - int(q_2[1]
        if pos_diff == 0:
```

```
            return
        # Check if already connected
        q_1_ind = self.vertices.index(q_1)
        q_2_ind = self.vertices.index(q_2)
        if (q_1_ind, q_2_ind) in self.edges or (q_2_ind, q_1_ind) in self.edges:
            return

        # Check if valid edge
        if self.check_collision_edge(q_1, q_2):
            return
        self.edges.add((q_1_ind, q_2_ind))

    def find_neighbors(self, node):
        vertices = np.asarray(self.vertices)
        vertices_2d = vertices[:, :2]
        node_2d = node[:2]
        distance = np.linalg.norm(vertices_2d - node_2d, axis=1)
        neighbor_ind = np.argsort(distance)[:self.num_neighbor]
        neighbors = [self.vertices[i] for i in neighbor_ind]
        return neighbors

    def connect_to_skeleton(self, pos):
        nodes_array = np.array(self.vertices)
        if len(nodes_array) == 0:
            return
        nodes_array_2d = nodes_array[:, :2]
        pos_2d = pos[:2]
        distances = np.linalg.norm(nodes_array_2d - pos_2d, axis=1)
        nearest_idx = np.argmin(distances)
        nearest_skel_point = tuple(nodes_array[nearest_idx])
        if self.check_collision_edge(pos, nearest_skel_point):
            return
        self.vertices.append(pos)
        pos_ind = self.vertices.index(pos)
        self.edges.add((pos_ind, nearest_idx))
        return

    def add_start_goal(self):
        self.connect_to_skeleton(self.start)
        self.connect_to_skeleton(self.goal)


    def swap_xy(self):
        vertices = [(y, x, theta) for x, y, theta in self.vertices]
        return vertices, self.edges

    def run(self):

        cspace_cpy = self.cspace.copy()
        while len(self.vertices) < self.num_sample:
            # Random sample angle
            angle = random.choice(self.angles)
            sample_space = cspace_cpy[angle]
            # Sample in free space
            free_space_indices = np.argwhere(sample_space == 0)
            free_ind = np.random.choice(len(free_space_indices))
            node = tuple(free_space_indices[free_ind])
            node = (node[0], node[1], angle)
            if node not in self.vertices:
                self.vertices.append(node)
```

```
        for node in self.vertices:
            neighbors = self.find_neighbors(node)
            for neighbor in neighbors:
                nodes = (node, neighbor)
                self.add_neighbors(nodes)


        self.add_start_goal()
        return self.swap_xy()
```

In [56]:
```
class Node:
    def __init__(self, x, y, theta, g=0, h=0, f=0, parent=None):
        self.x = x
        self.y = y
        self.theta = theta % 360
        self.g = g
        self.h = h
        self.f = f
        self.parent = parent
    def __eq__(self, other):
        return (self.x == other.x and self.y == other.y and self.theta == other.

    def __lt__(self, other):
        return self.f < other.f

    def key(self, angular_resolution):
        return f"{self.x}_{self.y}_{int(self.theta / angular_resolution)}"
```

In [57]:
```
class A_Star:
    def __init__(self, start, goal, cspace_maps, map_resolution=0.5, angular_res
        self.start = start
        self.goal = goal
        self.cspace_maps = cspace_maps
        self.map_resolution = map_resolution
        self.angular_resolution = angular_resolution
        self.open_list = []
        self.closed_set = set()
        self.map_size = cspace_maps[0].shape
        self.path = []
        self.total_steps = 0
        self.vertices = vertices
        self.edges = edges

    def init_open_list(self):
        start_node = Node(self.start[0], self.start[1], self.start[2])
        start_node.h = self.get_heuristic(start_node)
        start_node.f = start_node.g + start_node.h
        heapq.heappush(self.open_list, (start_node.f, start_node))

    def is_goal(self,node):
        return node.x == self.goal[0] and node.y == self.goal[1]

    def movement_cost(self, node, neighbor):
        # Translation Cost
        if node.x != neighbor.x or node.y != neighbor.y:
            rr,cc = line(node.x, node.y, neighbor.x, neighbor.y)
            return (len(rr) - 1)
```

```
        # Rotation Cost (Not Applied)
        else:
            angle_diff = min(abs(node.theta - neighbor.theta), 360 - abs(node.th
            steps = angle_diff / angular_resolution
            return steps

    def get_heuristic(self, node):
        '''
        Apply Euclidean distance to a node and goal.
        Args:
            node: input node

        Returns: heuristic value
        '''
        # Position Diff, Euclidean Distance
        pos_diff = np.sqrt((node.x - self.goal[0]) ** 2 + (node.y - self.goal[1]
        # Angle Diff
        angle_diff = min(abs(node.theta - self.goal[2]), 360 - abs(node.theta -
        return pos_diff + angle_diff

    def reconstruct_path(self,node):
        path = []
        total_steps = node.g
        while node:
            path.append((node.x, node.y, node.theta))
            node = node.parent
        path.reverse()
        return path, total_steps

    def gen_neighbors(self, node):
        neighbors = []
        # Find Neighbors
        node_ind = self.vertices.index((node.x, node.y, node.theta))
        for edge in self.edges:
            if node_ind in edge:
                neighbor_ind = edge[0] if edge[1] == node_ind else edge[1]
                neighbor = self.vertices[neighbor_ind]
                neighbors.append(Node(neighbor[0], neighbor[1], neighbor[2]))
        return neighbors

    def run(self):
        # Open List is not empty
        while self.open_list:
            current_node = heapq.heappop(self.open_list)[1]
            # Check Goal
            if self.is_goal(current_node):
                self.path, self.total_steps = self.reconstruct_path(current_node
                return
            # Add current node to closed set
            key = current_node.key(self.angular_resolution)
            self.closed_set.add(key)

            # Get Neighbor
            neighbors = self.gen_neighbors(current_node)
            for neighbor in neighbors:
                # Check if neighbor in closed list
                key = neighbor.key(self.angular_resolution)
                if key in self.closed_set:
                    continue
```

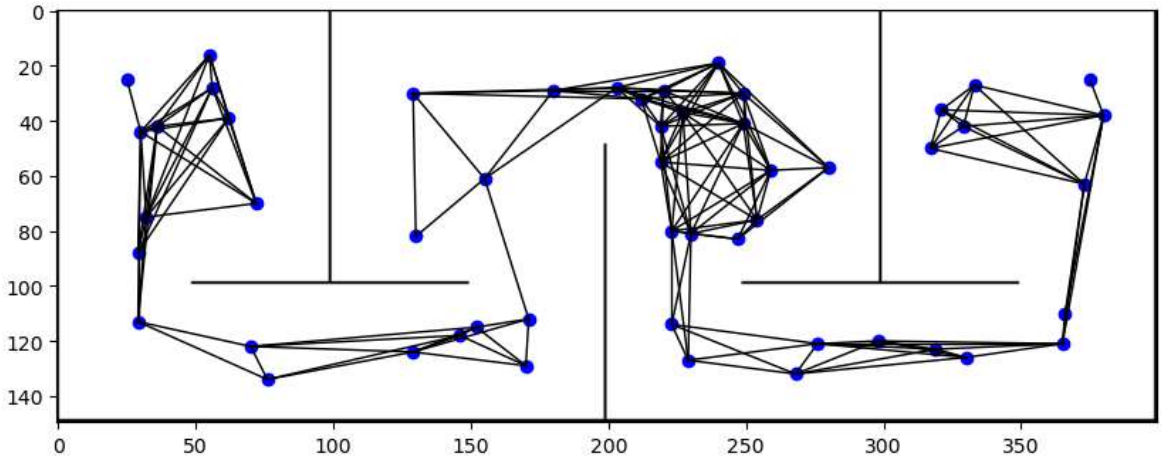
```
                # Compute g for neighbor
                tentative_g = current_node.g + self.movement_cost(current_node,

                # Update if neighbor in open list
                in_open_list = False
                for item in self.open_list:
                    if neighbor == item[1]:
                        in_open_list = True
                        if tentative_g < neighbor.g:
                            neighbor.g = tentative_g
                            neighbor.f = neighbor.g + neighbor.h
                            neighbor.parent = current_node
                        break
                # Add if neighbor not in open list
                if not in_open_list:
                    neighbor.g = tentative_g
                    neighbor.h = self.get_heuristic(neighbor)
                    neighbor.f = neighbor.g + neighbor.h
                    neighbor.parent = current_node
                    heapq.heappush(self.open_list, (neighbor.f, neighbor))
        return
```

In [32]:
```
map_resolution = 0.5
angular_resolution = 5
num_ori = 360 / angular_resolution
data = loadmat('cspaceMaps_python.mat')
angles = data['angles'][0]
map = data['mapMatrix']
cspace = {}
for index, angle in enumerate(angles):
    space = data['cspace'][0][index]
    cspace[angle] = space
```

PRM Sample:50

In [272…
```
start = (25, 25, 90)
goal = (25, 375, 0)
num_sample = 50
num_neighbor = 10
prm = PRM(start, goal, num_sample, num_neighbor, cspace, angles, angular_resolut
vertices,edges = prm.run()
```

In [276…
```
map_display = 1 - map
x_coords = [v[0] for v in vertices]
y_coords = [v[1] for v in vertices]

plt.figure(figsize=(10, 8))
plt.scatter(x_coords, y_coords, c='blue', label='Vertices')
for edge in edges:
    start, end = edge
    x = [vertices[start][0], vertices[end][0]]
    y = [vertices[start][1], vertices[end][1]]
    plt.plot(x, y, c='black', linestyle='-', linewidth=1)
plt.imshow(map_display, cmap='gray', origin='upper')
plt.savefig('PRM_sample_50.png')
plt.show()
```

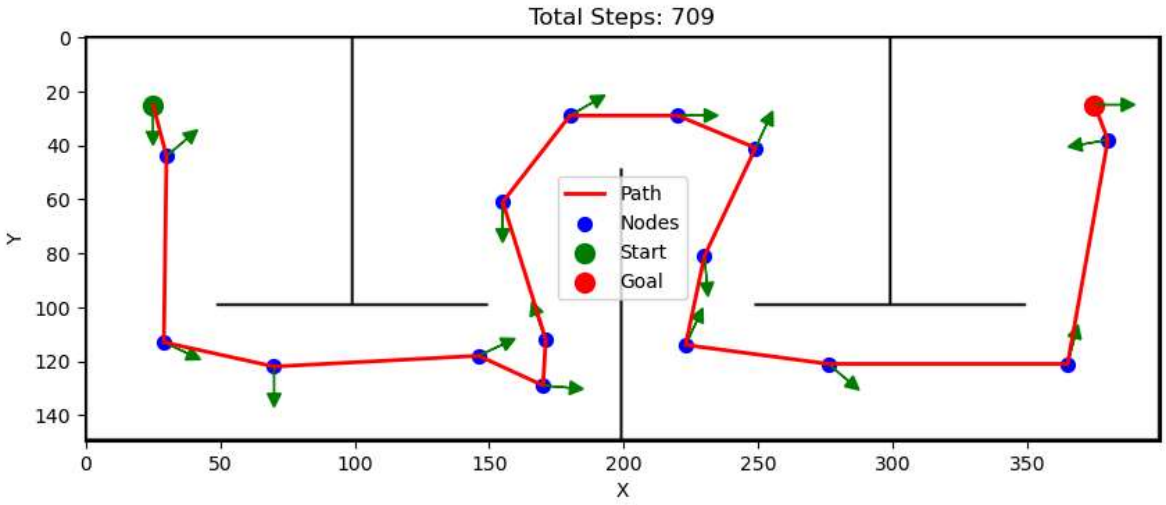


In [274…

```
start = (25, 25, 90)
goal = (375,25, 0)
map_resolution = 0.5
angular_resolution = 5

planner = A_Star(start, goal, cspace, map_resolution, angular_resolution, vertic
planner.init_open_list()
planner.run()
```

In [277…

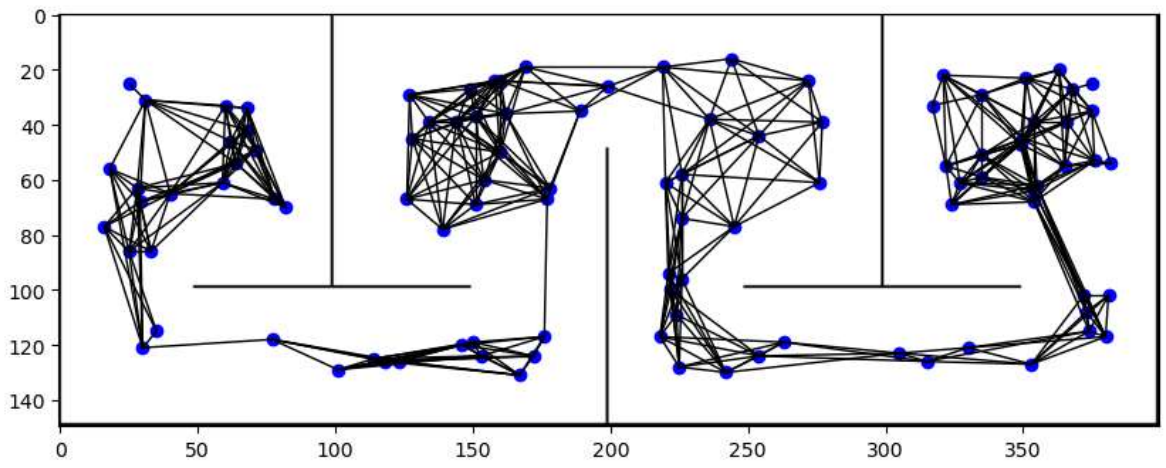
```
path = planner.path
path_x = [p[0] for p in path]
path_y = [p[1] for p in path]
path_theta = [p[2] for p in path]

plt.figure(figsize=(10, 8))
plt.imshow(map_display, cmap='gray', origin='upper')
plt.plot(path_x, path_y, 'r-', linewidth=2, label='Path')
plt.scatter(path_x, path_y, c='blue', s=50, label='Nodes')
plt.scatter(path_x[0], path_y[0], c='green', s=100, label='Start', marker='o')
plt.scatter(path_x[-1], path_y[-1], c='red', s=100, label='Goal', marker='o')
for x, y, theta in path:
    dx = np.cos(np.radians(theta)) * 10
    dy = np.sin(np.radians(theta)) * 10
    plt.arrow(x, y, dx, dy, head_width=5, head_length=5, fc='green', ec='green')
plt.title(f'Total Steps: {planner.total_steps}')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('PRM_50_path.png')
plt.show()
```



PRM Sample:100

In [297…
```
start = (25, 25, 90)
goal = (25, 375, 0)
num_sample = 100
num_neighbor = 10
prm = PRM(start, goal, num_sample, num_neighbor, cspace, angles, angular_resolut
vertices,edges = prm.run()
```

In [298…
```
map_display = 1 - map
x_coords = [v[0] for v in vertices]
y_coords = [v[1] for v in vertices]

plt.figure(figsize=(10, 8))
plt.scatter(x_coords, y_coords, c='blue', label='Vertices')
for edge in edges:
    start, end = edge
    x = [vertices[start][0], vertices[end][0]]
    y = [vertices[start][1], vertices[end][1]]
    plt.plot(x, y, c='black', linestyle='-', linewidth=1)
plt.imshow(map_display, cmap='gray', origin='upper')
plt.savefig('PRM_sample_100.png')
plt.show()
```

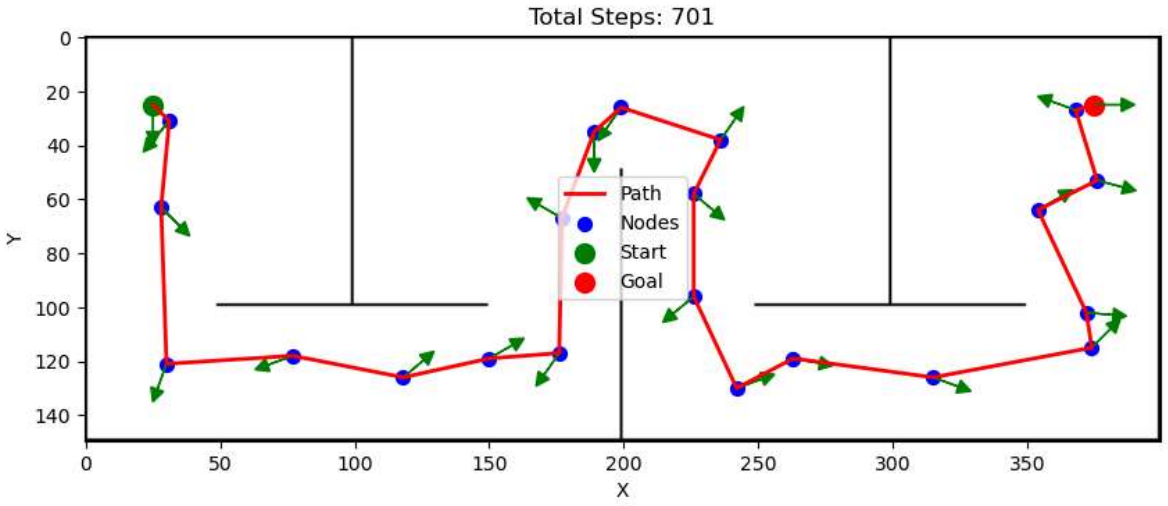


In [299…
```
start = (25, 25, 90)
goal = (375,25, 0)
map_resolution = 0.5
```

```
angular_resolution = 5

planner = A_Star(start, goal, cspace, map_resolution, angular_resolution, vertic
planner.init_open_list()
planner.run()
```

In [301…

```
path = planner.path
path_x = [p[0] for p in path]
path_y = [p[1] for p in path]
path_theta = [p[2] for p in path]

plt.figure(figsize=(10, 8))
plt.imshow(map_display, cmap='gray', origin='upper')
plt.plot(path_x, path_y, 'r-', linewidth=2, label='Path')
plt.scatter(path_x, path_y, c='blue', s=50, label='Nodes')
plt.scatter(path_x[0], path_y[0], c='green', s=100, label='Start', marker='o')
plt.scatter(path_x[-1], path_y[-1], c='red', s=100, label='Goal', marker='o')
for x, y, theta in path:
    dx = np.cos(np.radians(theta)) * 10
    dy = np.sin(np.radians(theta)) * 10
    plt.arrow(x, y, dx, dy, head_width=5, head_length=5, fc='green', ec='green')
plt.title(f'Total Steps: {planner.total_steps}')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('PRM_100_path.png')
plt.show()
```



PRM Sample:500

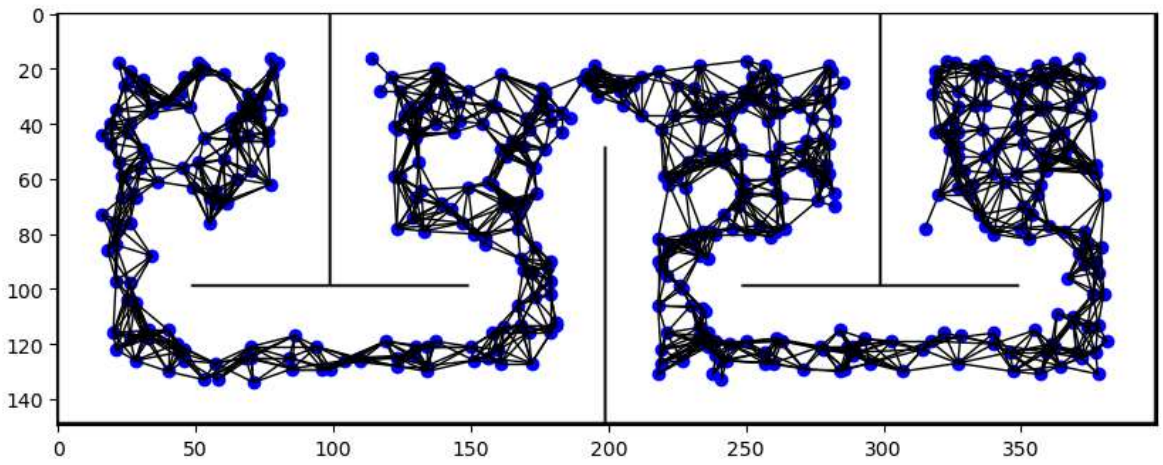
In [328…

```
start = (25, 25, 90)
goal = (25, 375, 0)
num_sample = 500
num_neighbor = 10
prm = PRM(start, goal, num_sample, num_neighbor, cspace, angles, angular_resolut
vertices,edges = prm.run()
```

In [332…

```
map_display = 1 - map
x_coords = [v[0] for v in vertices]
y_coords = [v[1] for v in vertices]

plt.figure(figsize=(10, 8))
```

```
plt.scatter(x_coords, y_coords, c='blue', label='Vertices')
for edge in edges:
    start, end = edge
    x = [vertices[start][0], vertices[end][0]]
    y = [vertices[start][1], vertices[end][1]]
    plt.plot(x, y, c='black', linestyle='-', linewidth=1)
plt.imshow(map_display, cmap='gray', origin='upper')
plt.savefig('PRM_sample_500.png')
plt.show()
```

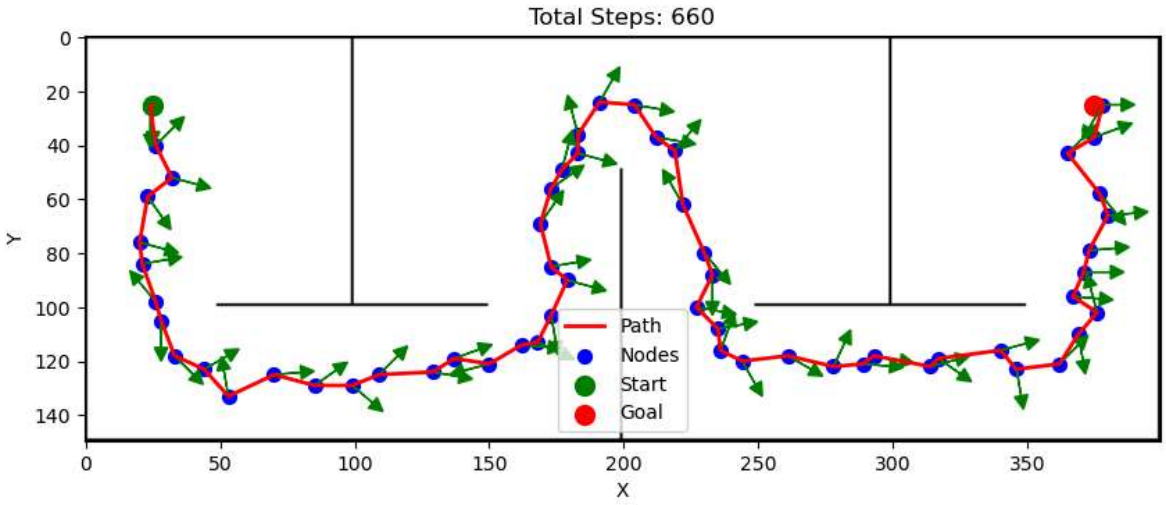


In [330…

```
start = (25, 25, 90)
goal = (375,25, 0)
map_resolution = 0.5
angular_resolution = 5

planner = A_Star(start, goal, cspace, map_resolution, angular_resolution, vertic
planner.init_open_list()
planner.run()
```

In [333…

```
path = planner.path
path_x = [p[0] for p in path]
path_y = [p[1] for p in path]
path_theta = [p[2] for p in path]

plt.figure(figsize=(10, 8))
plt.imshow(map_display, cmap='gray', origin='upper')
plt.plot(path_x, path_y, 'r-', linewidth=2, label='Path')
plt.scatter(path_x, path_y, c='blue', s=50, label='Nodes')
plt.scatter(path_x[0], path_y[0], c='green', s=100, label='Start', marker='o')
plt.scatter(path_x[-1], path_y[-1], c='red', s=100, label='Goal', marker='o')
for x, y, theta in path:
    dx = np.cos(np.radians(theta)) * 10
    dy = np.sin(np.radians(theta)) * 10
    plt.arrow(x, y, dx, dy, head_width=5, head_length=5, fc='green', ec='green')
plt.title(f'Total Steps: {planner.total_steps}')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('PRM_500_path.png')
plt.show()
```


Total Steps: 660
Path
Nodes
Start
Goal
X
Y
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300
350

In [ ]:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
import heapq
from skimage.draw import line
import random
```

RRT

In [267…
```
class RRT():
    def __init__(self, start, goal, num_sample, cspace_map,angles, angular_resol
        self.start = start
        self.goal = goal
        self.num_sample = num_sample
        self.cspace = cspace_map
        self.angles = angles
        self.angular_resolution = angular_resolution
        self.vertices = []
        self.edges = set()

    def find_nearest_vertex(self, node):
        nodes_array = np.array(self.vertices)
        if len(nodes_array) == 0:
            return
        nodes_array_2d = nodes_array[:, :2]
        node_2d = node[:2]
        distances = np.linalg.norm(nodes_array_2d - node_2d, axis=1)
        neighbor_ind = np.argmin(distances)
        neighbor = self.vertices[neighbor_ind]
        return neighbor

    def check_collision_node(self, node):
        node_x = node[0]
        node_y = node[1]
        node_theta = node[2]
        if self.cspace[node_theta][node_x][node_y] != 0:
            return True
        return False

    def check_collision_edge(self, q_1, q_2):
        # Check rotation
        angle_diff = (int(q_1[2]) - int(q_2[2]) + 360) % 360
        steps_to_rotate = angle_diff / self.angular_resolution
        if steps_to_rotate != 0:
            rotation_direction = self.angular_resolution if angle_diff <= 180 el
            temp_theta = int(q_1[2])
            while temp_theta != int(q_2[2]):
                temp_theta = (temp_theta + rotation_direction + 360) % 360
                # Rotation from q_1 to q_2 is blocked
                if self.check_collision_node((q_1[0], q_1[1], temp_theta)):
                    return True

        # Check connection
        rr,cc = line(q_1[0], q_1[1], q_2[0], q_2[1])
        for row, col in zip(rr,cc):
            if self.check_collision_node((row, col, q_2[2])):
```

```
                return True
        return False

    def find_new_config(self,q_1, q_2):
        # Check rotation
        angle_diff = (int(q_1[2]) - int(q_2[2]) + 360) % 360
        steps_to_rotate = angle_diff / self.angular_resolution
        if steps_to_rotate != 0:
            rotation_direction = self.angular_resolution if angle_diff <= 180 el
            temp_theta = int(q_1[2])
            while temp_theta != int(q_2[2]):
                temp_theta = (temp_theta + rotation_direction + 360) % 360
                # Rotation from q_1 to q_2 is blocked
                if self.check_collision_node((q_1[0], q_1[1], temp_theta)):
                    return None

        # Find new config with the longest distance
        rr,cc = line(q_1[0], q_1[1], q_2[0], q_2[1])
        new_config = None
        for row, col in zip(rr,cc):
            if self.check_collision_node((row, col, q_2[2])):
                break
            new_config = (row, col, q_2[2])
        return new_config

    def connect_to_skeleton(self, pos):
        nodes_array = np.array(self.vertices)
        if len(nodes_array) == 0:
            return
        nodes_array_2d = nodes_array[:, :2]
        pos_2d = pos[:2]
        distances = np.linalg.norm(nodes_array_2d - pos_2d, axis=1)
        nearest_idx = np.argmin(distances)
        nearest_skel_point = tuple(nodes_array[nearest_idx])
        if self.check_collision_edge(pos, nearest_skel_point):
            return
        self.vertices.append(pos)
        pos_ind = self.vertices.index(pos)
        self.edges.add((pos_ind, nearest_idx))
        return

    def add_goal(self):
        self.connect_to_skeleton(self.goal)

    def swap_xy(self):
        vertices = [(y, x, theta) for x, y, theta in self.vertices]
        return vertices, self.edges

    def run(self):
        # Add start to graph
        self.vertices.append(self.start)
        cspace_cpy = self.cspace.copy()
        while len(self.vertices) < self.num_sample:
            # Random sample angle
            angle = random.choice(self.angles)
            sample_space = cspace_cpy[angle]
            # Sample in free space
            free_space_indices = np.argwhere(sample_space == 0)
            free_ind = np.random.choice(len(free_space_indices))
            node = tuple(free_space_indices[free_ind])
```

```
            node = (node[0], node[1], angle)
            if any(v[0] == node[0] and v[1] == node[1] for v in self.vertices):
                continue

            # Find Neighbor (Nearest)
            neighbor = self.find_nearest_vertex(node)
            # Can Connect
            if not self.check_collision_edge(neighbor, node):
                self.vertices.append(node)
                node_ind = self.vertices.index(node)
                neighbor_ind = self.vertices.index(neighbor)
                self.edges.add((neighbor_ind, node_ind))
                continue
            # Can't Connect, find new config
            '''node = self.find_new_config(neighbor, node)
            if node is None:
                continue
            self.vertices.append(node)
            node_ind = self.vertices.index(node)
            neighbor_ind = self.vertices.index(neighbor)
            self.edges.add((neighbor_ind, node_ind))'''
        self.add_goal()
        return self.swap_xy()
```

In [425…

```
class Node:
    def __init__(self, x, y, theta, g=0, h=0, f=0, parent=None):
        self.x = x
        self.y = y
        self.theta = theta % 360
        self.g = g
        self.h = h
        self.f = f
        self.parent = parent
    def __eq__(self, other):
        return (self.x == other.x and self.y == other.y and self.theta == other.

    def __lt__(self, other):
        return self.f < other.f

    def key(self, angular_resolution):
        return f"{self.x}_{self.y}_{int(self.theta / angular_resolution)}"
```

In [426…

```
class A_Star:
    def __init__(self, start, goal, cspace_maps, map_resolution=0.5, angular_res
        self.start = start
        self.goal = goal
        self.cspace_maps = cspace_maps
        self.map_resolution = map_resolution
        self.angular_resolution = angular_resolution
        self.open_list = []
        self.closed_set = set()
        self.map_size = cspace_maps[0].shape
        self.path = []
        self.total_steps = 0
        self.vertices = vertices
        self.edges = edges

    def init_open_list(self):
        start_node = Node(self.start[0], self.start[1], self.start[2])
```

```
        start_node.h = self.get_heuristic(start_node)
        start_node.f = start_node.g + start_node.h
        heapq.heappush(self.open_list, (start_node.f, start_node))

    def is_goal(self,node):
        return node.x == self.goal[0] and node.y == self.goal[1]

    def movement_cost(self, node, neighbor):
        # Translation Cost
        if node.x != neighbor.x or node.y != neighbor.y:
            rr,cc = line(node.x, node.y, neighbor.x, neighbor.y)
            return (len(rr) - 1)
        # Rotation Cost (Not Applied)
        else:
            angle_diff = min(abs(node.theta - neighbor.theta), 360 - abs(node.th
            steps = angle_diff / angular_resolution
            return steps

    def get_heuristic(self, node):
        '''
        Apply Euclidean distance to a node and goal.
        Args:
            node: input node

        Returns: heuristic value
        '''
        # Position Diff, Euclidean Distance
        pos_diff = np.sqrt((node.x - self.goal[0]) ** 2 + (node.y - self.goal[1]
        # Angle Diff
        angle_diff = min(abs(node.theta - self.goal[2]), 360 - abs(node.theta -
        return pos_diff + angle_diff

    def reconstruct_path(self,node):
        path = []
        total_steps = node.g
        while node:
            path.append((node.x, node.y, node.theta))
            node = node.parent
        path.reverse()
        return path, total_steps

    def gen_neighbors(self, node):
        neighbors = []
        # Find Neighbors
        node_ind = self.vertices.index((node.x, node.y, node.theta))
        for edge in self.edges:
            if node_ind in edge:
                neighbor_ind = edge[0] if edge[1] == node_ind else edge[1]
                neighbor = self.vertices[neighbor_ind]
                neighbors.append(Node(neighbor[0], neighbor[1], neighbor[2]))
        return neighbors

    def run(self):
        # Open List is not empty
        while self.open_list:
            current_node = heapq.heappop(self.open_list)[1]
            # Check Goal
            if self.is_goal(current_node):
                self.path, self.total_steps = self.reconstruct_path(current_node
                return
```

```
            # Add current node to closed set
            key = current_node.key(self.angular_resolution)
            self.closed_set.add(key)

            # Get Neighbor
            neighbors = self.gen_neighbors(current_node)
            for neighbor in neighbors:
                # Check if neighbor in closed list
                key = neighbor.key(self.angular_resolution)
                if key in self.closed_set:
                    continue

                # Compute g for neighbor
                tentative_g = current_node.g + self.movement_cost(current_node,

                # Update if neighbor in open list
                in_open_list = False
                for item in self.open_list:
                    if neighbor == item[1]:
                        in_open_list = True
                        if tentative_g < neighbor.g:
                            neighbor.g = tentative_g
                            neighbor.f = neighbor.g + neighbor.h
                            neighbor.parent = current_node
                        break
                # Add if neighbor not in open list
                if not in_open_list:
                    neighbor.g = tentative_g
                    neighbor.h = self.get_heuristic(neighbor)
                    neighbor.f = neighbor.g + neighbor.h
                    neighbor.parent = current_node
                    heapq.heappush(self.open_list, (neighbor.f, neighbor))
        return
```

In [268…

```
map_resolution = 0.5
angular_resolution = 5
num_ori = 360 / angular_resolution
data = loadmat('cspaceMaps_python.mat')
angles = data['angles'][0]
map = data['mapMatrix']
cspace = {}
for index, angle in enumerate(angles):
    space = data['cspace'][0][index]
    cspace[angle] = space
```

RRT Sample:500

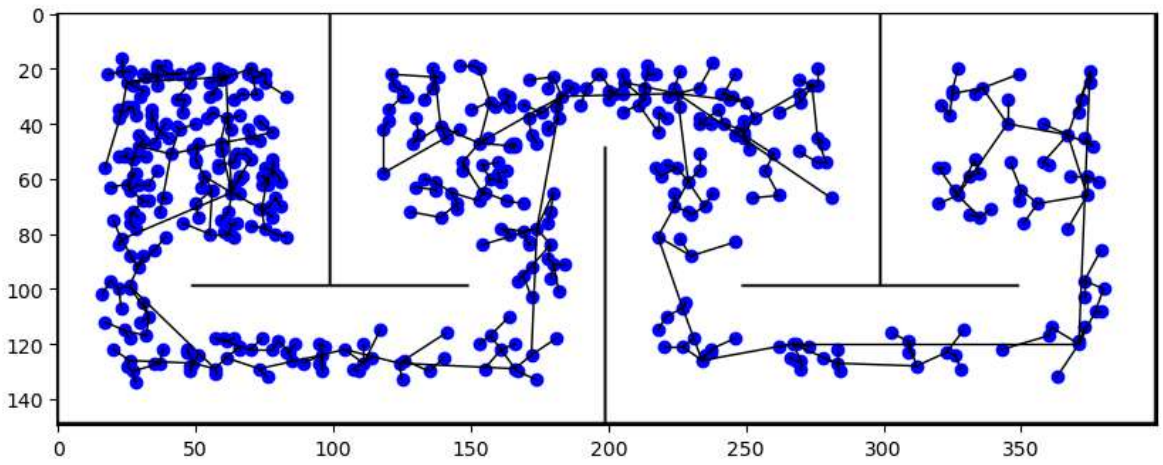
In [443…

```
start = (25, 25, 90)
goal = (25, 375, 0)
num_sample = 500
rrt = RRT(start, goal, num_sample, cspace, angles, angular_resolution)
vertices,edges = rrt.run()
```

In [444…

```
map_display = 1 - map
x_coords = [v[0] for v in vertices]
y_coords = [v[1] for v in vertices]

plt.figure(figsize=(10, 8))
```

```
plt.scatter(x_coords, y_coords, c='blue', label='Vertices')
for edge in edges:
    start, end = edge
    x = [vertices[start][0], vertices[end][0]]
    y = [vertices[start][1], vertices[end][1]]
    plt.plot(x, y, c='black', linestyle='-', linewidth=1)
plt.imshow(map_display, cmap='gray', origin='upper')
#plt.savefig('PRT_sample_500_unguide.png')
plt.show()
```

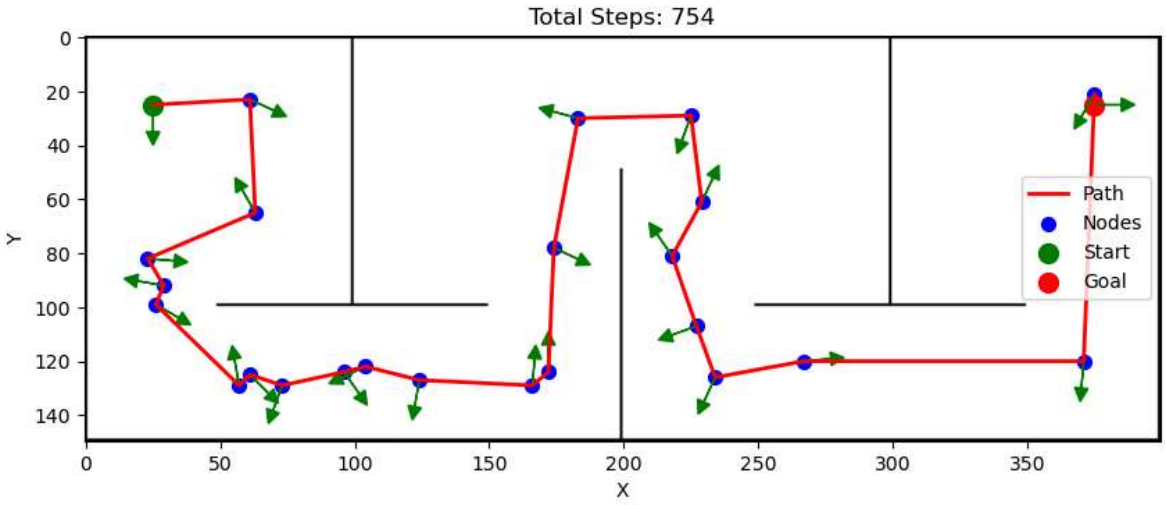


In [445…

```
start = (25, 25, 90)
goal = (375,25, 0)
map_resolution = 0.5
angular_resolution = 5

planner = A_Star(start, goal, cspace, map_resolution, angular_resolution, vertic
planner.init_open_list()
planner.run()
```

In [446…

```
path = planner.path
path_x = [p[0] for p in path]
path_y = [p[1] for p in path]
path_theta = [p[2] for p in path]

plt.figure(figsize=(10, 8))
plt.imshow(map_display, cmap='gray', origin='upper')
plt.plot(path_x, path_y, 'r-', linewidth=2, label='Path')
plt.scatter(path_x, path_y, c='blue', s=50, label='Nodes')
plt.scatter(path_x[0], path_y[0], c='green', s=100, label='Start', marker='o')
plt.scatter(path_x[-1], path_y[-1], c='red', s=100, label='Goal', marker='o')
for x, y, theta in path:
    dx = np.cos(np.radians(theta)) * 10
    dy = np.sin(np.radians(theta)) * 10
    plt.arrow(x, y, dx, dy, head_width=5, head_length=5, fc='green', ec='green')
plt.title(f'Total Steps: {planner.total_steps}')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
#plt.savefig('PRT_500_path.png')
plt.show()
```


Total Steps: 754
Path
Nodes
Start
Goal
X
Y
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300
350

In [ ]:

In [77]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
import heapq
from skimage.draw import line
import random
```

RRT Guided

In [139…
```
class Node:
    def __init__(self, x, y, theta, g=0, h=0, f=0, parent=None):
        self.x = x
        self.y = y
        self.theta = theta % 360
        self.g = g
        self.h = h
        self.f = f
        self.parent = parent
    def __eq__(self, other):
        return (self.x == other.x and self.y == other.y and self.theta == other.

    def __lt__(self, other):
        return self.f < other.f

    def key(self, angular_resolution):
        return f"{self.x}_{self.y}_{int(self.theta / angular_resolution)}"
```

In [152…
```
class A_Star:
    def __init__(self, start, goal, cspace_maps, map_resolution=0.5, angular_res
        self.start = start
        self.goal = goal
        self.cspace_maps = cspace_maps
        self.map_resolution = map_resolution
        self.angular_resolution = angular_resolution
        self.open_list = []
        self.closed_set = set()
        self.map_size = cspace_maps[0].shape
        self.path = []
        self.total_steps = 0
        self.vertices = vertices
        self.edges = edges

    def init_open_list(self):
        start_node = Node(self.start[0], self.start[1], self.start[2])
        start_node.h = self.get_heuristic(start_node)
        start_node.f = start_node.g + start_node.h
        heapq.heappush(self.open_list, (start_node.f, start_node))

    def is_goal(self,node):
        return node.x == self.goal[0] and node.y == self.goal[1]


    def movement_cost(self, node, neighbor):
        # Translation Cost
        if node.x != neighbor.x or node.y != neighbor.y:
            rr,cc = line(node.x, node.y, neighbor.x, neighbor.y)
```

```
            return (len(rr) - 1)
        # Rotation Cost (Not Applied)
        else:
            angle_diff = min(abs(node.theta - neighbor.theta), 360 - abs(node.th
            steps = angle_diff / angular_resolution
            return steps

    def get_heuristic(self, node):
        '''
        Apply Euclidean distance to a node and goal.
        Args:
            node: input node

        Returns: heuristic value
        '''
        # Position Diff, Euclidean Distance
        pos_diff = np.sqrt((node.x - self.goal[0]) ** 2 + (node.y - self.goal[1]
        # Angle Diff
        angle_diff = min(abs(node.theta - self.goal[2]), 360 - abs(node.theta -
        return pos_diff + angle_diff

    def reconstruct_path(self,node):
        path = []
        total_steps = node.g
        while node:
            path.append((node.x, node.y, node.theta))
            node = node.parent
        path.reverse()
        return path, total_steps

    def gen_neighbors(self, node):
        neighbors = []
        # Find Neighbors
        node_ind = self.vertices.index((node.x, node.y, node.theta))
        for edge in self.edges:
            if node_ind in edge:
                neighbor_ind = edge[0] if edge[1] == node_ind else edge[1]
                neighbor = self.vertices[neighbor_ind]
                neighbors.append(Node(neighbor[0], neighbor[1], neighbor[2]))
        return neighbors

    def run(self):
        # Open List is not empty
        while self.open_list:
            current_node = heapq.heappop(self.open_list)[1]
            # Check Goal
            if self.is_goal(current_node):
                self.path, self.total_steps = self.reconstruct_path(current_node
                return
            # Add current node to closed set
            key = current_node.key(self.angular_resolution)
            self.closed_set.add(key)

            # Get Neighbor
            neighbors = self.gen_neighbors(current_node)
            for neighbor in neighbors:
                # Check if neighbor in closed list
                key = neighbor.key(self.angular_resolution)
                if key in self.closed_set:
                    continue
```

```
                # Compute g for neighbor
                tentative_g = current_node.g + self.movement_cost(current_node,

                # Update if neighbor in open list
                in_open_list = False
                for item in self.open_list:
                    if neighbor == item[1]:
                        in_open_list = True
                        if tentative_g < neighbor.g:
                            neighbor.g = tentative_g
                            neighbor.f = neighbor.g + neighbor.h
                            neighbor.parent = current_node
                        break
                # Add if neighbor not in open list
                if not in_open_list:
                    neighbor.g = tentative_g
                    neighbor.h = self.get_heuristic(neighbor)
                    neighbor.f = neighbor.g + neighbor.h
                    neighbor.parent = current_node
                    heapq.heappush(self.open_list, (neighbor.f, neighbor))
        return
```

In [227…

```
class Guided_RRT():
    def __init__(self, start, goal, num_sample, cspace_map,angles, angular_resol
        self.start = start
        self.goal = goal
        self.num_sample = num_sample
        self.cspace = cspace_map
        self.angles = angles
        self.angular_resolution = angular_resolution
        self.vertices = []
        self.edges = set()
        self.intermediate_goals = intermediate_goals if intermediate_goals else

    def find_nearest_vertex(self, node):
        nodes_array = np.array(self.vertices)
        if len(nodes_array) == 0:
            return
        nodes_array_2d = nodes_array[:, :2]
        node_2d = node[:2]
        distances = np.linalg.norm(nodes_array_2d - node_2d, axis=1)
        neighbor_ind = np.argmin(distances)
        neighbor = self.vertices[neighbor_ind]
        return neighbor

    def check_collision_node(self, node):
        node_x = node[0]
        node_y = node[1]
        node_theta = node[2]
        if self.cspace[node_theta][node_x][node_y] != 0:
            return True
        return False

    def check_collision_edge(self, q_1, q_2):
        # Check rotation
        angle_diff = (int(q_1[2]) - int(q_2[2]) + 360) % 360
        steps_to_rotate = angle_diff / self.angular_resolution
        if steps_to_rotate != 0:
            rotation_direction = self.angular_resolution if angle_diff <= 180 el
```

```
            temp_theta = int(q_1[2])
            while temp_theta != int(q_2[2]):
                temp_theta = (temp_theta + rotation_direction + 360) % 360
                # Rotation from q_1 to q_2 is blocked
                if self.check_collision_node((q_1[0], q_1[1], temp_theta)):
                    return True

        # Check connection
        rr,cc = line(q_1[0], q_1[1], q_2[0], q_2[1])
        for row, col in zip(rr,cc):
            if self.check_collision_node((row, col, q_2[2])):
                return True
        return False

    def find_new_config(self,q_1, q_2):
        # Check rotation
        angle_diff = (int(q_1[2]) - int(q_2[2]) + 360) % 360
        steps_to_rotate = angle_diff / self.angular_resolution
        if steps_to_rotate != 0:
            rotation_direction = self.angular_resolution if angle_diff <= 180 el
            temp_theta = int(q_1[2])
            while temp_theta != int(q_2[2]):
                temp_theta = (temp_theta + rotation_direction + 360) % 360
                # Rotation from q_1 to q_2 is blocked
                if self.check_collision_node((q_1[0], q_1[1], temp_theta)):
                    return None

        # Find new config with the longest distance
        rr,cc = line(q_1[0], q_1[1], q_2[0], q_2[1])
        new_config = None
        for row, col in zip(rr,cc):
            if self.check_collision_node((row, col, q_2[2])):
                break
            new_config = (row, col, q_2[2])
        return new_config

    def connect_to_skeleton(self, pos):
        nodes_array = np.array(self.vertices)
        if len(nodes_array) == 0:
            return
        nodes_array_2d = nodes_array[:, :2]
        pos_2d = pos[:2]
        distances = np.linalg.norm(nodes_array_2d - pos_2d, axis=1)
        nearest_idx = np.argmin(distances)
        nearest_skel_point = tuple(nodes_array[nearest_idx])
        if self.check_collision_edge(pos, nearest_skel_point):
            return
        self.vertices.append(pos)
        pos_ind = self.vertices.index(pos)
        self.edges.add((pos_ind, nearest_idx))
        return

    def add_goal(self):
        self.connect_to_skeleton(self.goal)

    def swap_xy(self):
        vertices = [(y, x, theta) for x, y, theta in self.vertices]
        return vertices, self.edges

    def distance_to_goal(self, node, goal):
```

```
        return np.sqrt((node[0] - goal[0]) ** 2 + (node[1] - goal[1]) ** 2)

    def add_remain(self):
        cspace_cpy = self.cspace.copy()
        while len(self.vertices) < self.num_sample:
            # Random sample angle
            angle = random.choice(self.angles)
            sample_space = cspace_cpy[angle]
            # Sample in free space
            free_space_indices = np.argwhere(sample_space == 0)
            free_ind = np.random.choice(len(free_space_indices))
            node = tuple(free_space_indices[free_ind])
            node = (node[0], node[1], angle)
            if any(v[0] == node[0] and v[1] == node[1] for v in self.vertices):
                continue

            # Find Neighbor (Nearest)
            neighbor = self.find_nearest_vertex(node)
            # Can Connect
            if not self.check_collision_edge(neighbor, node):
                self.vertices.append(node)
                node_ind = self.vertices.index(node)
                neighbor_ind = self.vertices.index(neighbor)
                self.edges.add((neighbor_ind, node_ind))
                continue


    def run(self):
        # Add start to graph
        self.vertices.append(self.start)
        cspace_cpy = self.cspace.copy()

        for intermediate_goal in self.intermediate_goals:
            count = 0
            while count <= 14:
                node = None
                if random.random() < 0.5:  # 30% 的概率偏向当前中间目标
                    angle = random.choice(self.angles)
                    node = (intermediate_goal[0], intermediate_goal[1], angle)
                else:
                    # Random sample angle
                    angle = random.choice(self.angles)
                    sample_space = cspace_cpy[angle]
                    # Sample in free space
                    free_space_indices = np.argwhere(sample_space == 0)
                    free_ind = np.random.choice(len(free_space_indices))
                    node = tuple(free_space_indices[free_ind])
                    node = (node[0], node[1], angle)
                if any(v[0] == node[0] and v[1] == node[1] for v in self.vertice
                    continue

                # Find Neighbor (Nearest)
                neighbor = self.find_nearest_vertex(node)
                # Can Connect
                if not self.check_collision_edge(neighbor, node):
                    self.vertices.append(node)
                    node_ind = self.vertices.index(node)
                    neighbor_ind = self.vertices.index(neighbor)
                    self.edges.add((neighbor_ind, node_ind))
                    count += 1
```

```
                '''if self.distance_to_goal(node, intermediate_goal) < 5:  # 距离
                    break'''


        self.add_goal()
        return self.swap_xy()
```

In [186…

```
map_resolution = 0.5
angular_resolution = 5
num_ori = 360 / angular_resolution
data = loadmat('cspaceMaps_python.mat')
angles = data['angles'][0]
map = data['mapMatrix']
cspace = {}
for index, angle in enumerate(angles):
    space = data['cspace'][0][index]
    cspace[angle] = space
```

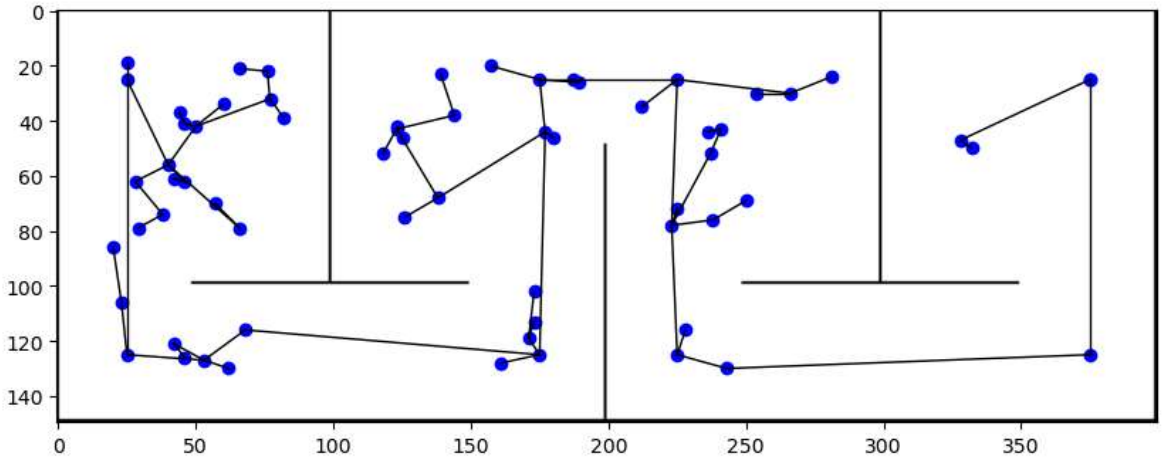
RRT Guided Sample:50

In [211…

```
start = (25, 25, 90)
goal = (25, 375, 0)
num_sample = 50
intermediate_goals = [
    (125,25,0),
    (125,175,0),
    (25, 175, 0),
    (25, 225, 0),
    (125,225, 0),
    (125,375,0),
    (25,375,0),
]
rrt = Guided_RRT(start, goal, num_sample, cspace, angles, angular_resolution, in
vertices,edges = rrt.run()
```

In [213…

```
map_display = 1 - map
x_coords = [v[0] for v in vertices]
y_coords = [v[1] for v in vertices]

plt.figure(figsize=(10, 8))
plt.scatter(x_coords, y_coords, c='blue', label='Vertices')
for edge in edges:
    start, end = edge
    x = [vertices[start][0], vertices[end][0]]
    y = [vertices[start][1], vertices[end][1]]
    plt.plot(x, y, c='black', linestyle='-', linewidth=1)
plt.imshow(map_display, cmap='gray', origin='upper')
plt.savefig('RRT_sample_50_guided.png')
plt.show()
```

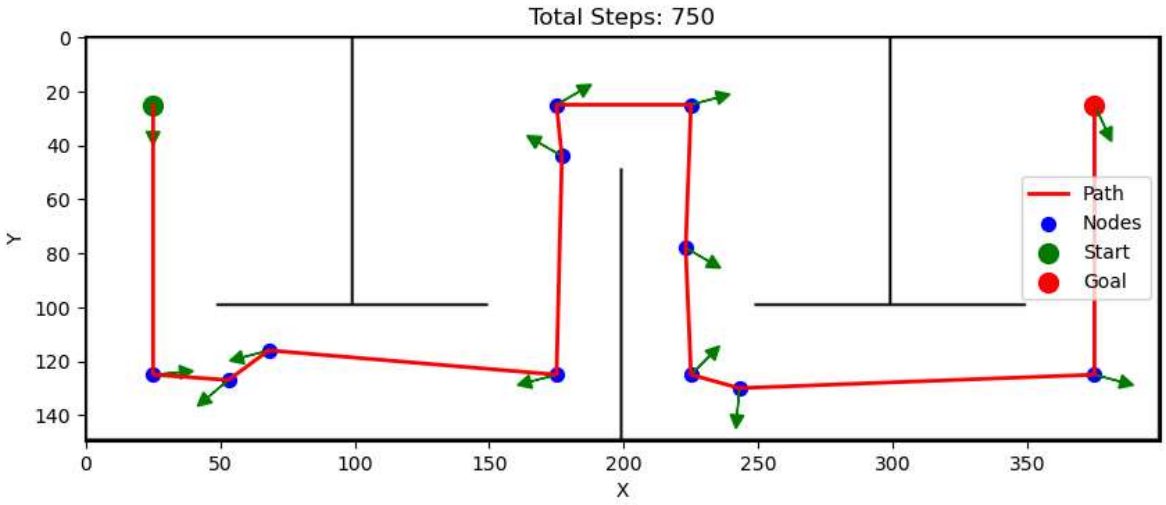


In [214…

```
start = (25, 25, 90)
goal = (375,25, 0)
map_resolution = 0.5
angular_resolution = 5

planner = A_Star(start, goal, cspace, map_resolution, angular_resolution, vertic
planner.init_open_list()
planner.run()
```

In [215…

```
path = planner.path
path_x = [p[0] for p in path]
path_y = [p[1] for p in path]
path_theta = [p[2] for p in path]

plt.figure(figsize=(10, 8))
plt.imshow(map_display, cmap='gray', origin='upper')
plt.plot(path_x, path_y, 'r-', linewidth=2, label='Path')
plt.scatter(path_x, path_y, c='blue', s=50, label='Nodes')
plt.scatter(path_x[0], path_y[0], c='green', s=100, label='Start', marker='o')
plt.scatter(path_x[-1], path_y[-1], c='red', s=100, label='Goal', marker='o')
for x, y, theta in path:
    dx = np.cos(np.radians(theta)) * 10
    dy = np.sin(np.radians(theta)) * 10
    plt.arrow(x, y, dx, dy, head_width=5, head_length=5, fc='green', ec='green')

plt.title(f'Total Steps: {planner.total_steps}')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
#plt.savefig('RRT_50_path.png')
plt.show()
```



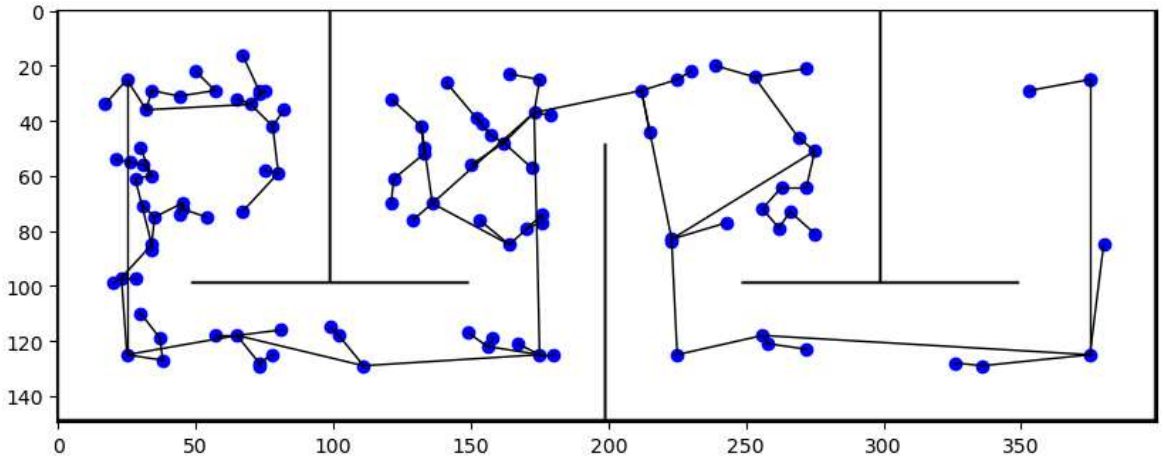
RRT Guided Sample:100

In [266…

```
start = (25, 25, 90)
goal = (25, 375, 0)
num_sample = 100
intermediate_goals = [
    (125,25,0),
    (125,175,0),
    (25, 175, 0),
    (25, 225, 0),
    (125,225, 0),
    (125,375,0),
    (25,375,0),
]
rrt = Guided_RRT(start, goal, num_sample, cspace, angles, angular_resolution, in
vertices,edges = rrt.run()
```

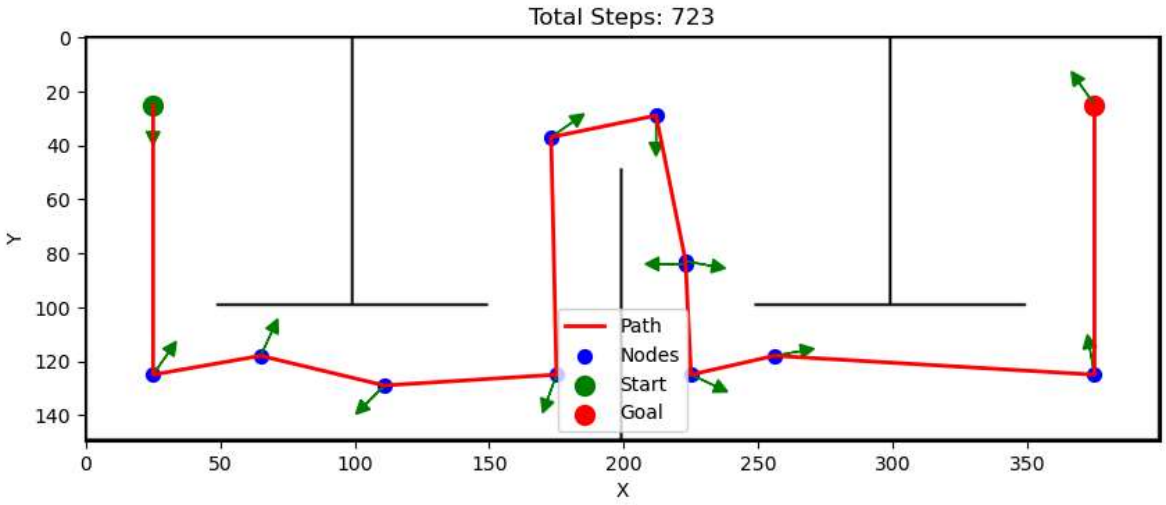
In [270…

```
map_display = 1 - map
x_coords = [v[0] for v in vertices]
y_coords = [v[1] for v in vertices]

plt.figure(figsize=(10, 8))
plt.scatter(x_coords, y_coords, c='blue', label='Vertices')
for edge in edges:
    start, end = edge
    x = [vertices[start][0], vertices[end][0]]
    y = [vertices[start][1], vertices[end][1]]
    plt.plot(x, y, c='black', linestyle='-', linewidth=1)
plt.imshow(map_display, cmap='gray', origin='upper')
plt.savefig('RRT_sample_100_guided.png')
plt.show()
```



In [268…
```
start = (25, 25, 90)
goal = (375,25, 0)
map_resolution = 0.5
angular_resolution = 5

planner = A_Star(start, goal, cspace, map_resolution, angular_resolution, vertic
planner.init_open_list()
planner.run()
```

In [271…
```
path = planner.path
path_x = [p[0] for p in path]
path_y = [p[1] for p in path]
path_theta = [p[2] for p in path]

plt.figure(figsize=(10, 8))
plt.imshow(map_display, cmap='gray', origin='upper')
plt.plot(path_x, path_y, 'r-', linewidth=2, label='Path')
plt.scatter(path_x, path_y, c='blue', s=50, label='Nodes')
plt.scatter(path_x[0], path_y[0], c='green', s=100, label='Start', marker='o')
plt.scatter(path_x[-1], path_y[-1], c='red', s=100, label='Goal', marker='o')
for x, y, theta in path:
    dx = np.cos(np.radians(theta)) * 10
    dy = np.sin(np.radians(theta)) * 10
    plt.arrow(x, y, dx, dy, head_width=5, head_length=5, fc='green', ec='green')

plt.title(f'Total Steps: {planner.total_steps}')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('RRT_100_path.png')
plt.show()
```


Total Steps: 723
Path
Nodes
Start
Goal
X
Y
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300
350

In [ ]: